Parallel Computer Architecture
Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 04
Lecture - 21

Lec 21: Coherence related Conditions

Hello everyone. We are doing the module on Cache Coherence. We will continue with the lecture about understanding cache coherence and this particular lecture is going to concentrate on coherence related conditions. In the previous lecture, we discussed about the concept of serialization that is when multiple processes run on different multiprocessors, each process has to follow its program order and when the read write request eventually reach the memory, they will reach in some order. Actually it will be a some arbitrary interleaving among these read write accesses and this interleaving could be distinct in every run which you do in the multiprocessor. However, it has to follow some hypothetical total order. Right.

So, every process will follow a program order and there will be an arbitrary interleaving among the read write accesses of multiple processes accesses. So when we are looking at a coherent view of the memory, what do we mean? We mean to say that any read done by a process should return the most recently written value to that location. Okay. Now this recently written value will be definitely the one in its own program order if this is the only program running or if it could be the value written by another process running on another processor. So and this previously written value will be in order in this hypothetical total order or the serialized order which we have constructed. Right.

So in a uniprocessor, the most recently written value is the one in your own program order, but it becomes slightly involving or complicated when we go to a multiprocessor setup. Right. So. Okay. So this concept is very easy in a uniprocessor, but when we go to multiple processes, we need to understand how does the memory system behave and how are the values transferred from one to the other. So to formalize this, we are going to divide this task into two aspects. One is called coherence and the other is called consistency. Okay. So memory coherence essentially defines that what values will a read return.

So when a process reads a value, what actually is the value it is going to get? So this is handled by coherence and it defines the values with respect to a given location. So I am talking of a location x and if I read location x, what exactly is the value I am going to get

is handled by the concept of coherence. And the concept of consistency talks about when a written value will be returned. That is when I read a variable x, actually what is the time duration after the write that I am able to read this value. So coherence is going to define what values will be returned and consistency defines when a written value will be read by another process. Okay.

So to understand what is a coherent memory system, we are going to look at three conditions. The first is preserving program order, second is having a coherent view of the memory and third is write serialization. So we will see them one by one. The first is preserving coherent program order. Okay. So this is with respect to a single process.

So imagine a uniprocessor setup or in a multiprocessor setup assume a single processor running on a single, single process running on a single processor. Okay. So here if this process reads a memory location x. Okay. And now this read is following a write by the same processor, then the particular read should get the same value what is written by the same process because the process is going to follow its own program order. Okay. So I will illustrate it here. Right. So I will take three processors P1, P2, P3 or you can also call them as three processes running on these three processors. So these are three processors running the processes and I will just take a variable x, suppose it is initialized to 0.

After a while process P1 does a write of x equal to 2. Right. So x is made to by process P1 and we also have another additional assumption here that during this duration before P1 does a read on x. Okay. So P1 does a write x equal to 2 and later P1 does a read of x. So what value is x going to return here? So actually the value return will be x equal to 2. Now I will get this x equal to 2 provided the P2 and P3 processes do not change it. Right?

So x is my shared variable and here during this time and during this time x is not written. Okay. So they are not written at all. So I will go back to the definition. It says that a read by a processor P to a location x, so read by P to a location x that follows a write by P to x with no writes to x by another processor occurring between the write and the read done by the process will always return the value written by P. Okay. So here the variable x is a shared variable if P1 changes x to 2 and P2 and P3 do not access x.

So subsequently when P1 reads the value of x it will get the value as 2. So this is called preserving the program order. So program order with respect to P1 is maintained in this scenario. Okay. Second condition is coherent view of memory. Now here we are going to talk about reads and writes across two different processes.

Previous example was reads and writes across a single process. Now we are going to

say that one process writes and the other reads. So in this I am going to say if P2 changes x and then P1 reads it, what are the conditions under which P1 will get the value which P2 wrote. Okay. So first we will illustrate and then we will read out the definition. Okay. So I will take the same scenario but this time x will be modified by P2. Okay.

So maybe initially x was equal to 2 here and later P2 did write of x equal to 5. So P2 wrote x equal to 5 after some time and P1 is going to read x. So what value will P1 get? Will it get the value of 2 or will it get the value of 5? So that is the question. So the rule which we are discussing says that after this write is done and before this read is done here the others do not use x or do not change x. Right. So x is not used by P3, x is not used by P1 in this duration and later eventually this value will go here provided we have a long enough time spent after writing x equal to 5 and reading x.

So in this case the value I will get is equal to 5 and not 2. Okay. So we will go back to the definition and then understand this example. Okay. So here it says that read by a processor to a location x, read by a processor to a location x that follows a write by another processor to that same location x, if these two are separated sufficiently in time then the changes by the other processor will reach this current processor. Okay. So that is the coherent view of memory.

Changes by another processor are seen by changes in the remaining processors in the system. Okay. So read by a processor to x that follows a write but in this case by another processor. The previous program order condition said the same processor whereas here the write is done by another processor. Okay. And the other condition is they are sufficiently separated in time. So there has to be enough time for the write of the other processor to reach the system level or other processor should be able to see this change and only then we will be able to access the current value. Okay. So that is the meaning of coherent view of memory. Okay.

Now the third condition is write serialization. Now this says that if I am doing two consecutive writes to the same location be it by the same processor or be these writes done by different processors. But these writes should be seen in the same order by everybody in the system. By everybody in the system meaning the main memory should see the same order, any other processor should also see the same order. So if I am writing a value equal to 1 to some variable x then I write a value equal to 2 to the same variable x. So what is the order of writes? First x becomes 1 then x becomes 2.

So there should be no other process in the system which will see x equal to 2 first and then it will see x equal to 1. So this will violate the write serialization property. Okay. So we will again illustrate this using a similar example. Right. So we have again process P1,

process P2. So process P1 writes x equal to 1.

Process P2 writes x equal to 2. Now we are not discussing about what is the time separation between them. We are just saying probably they happen in parallel somehow and when they happen they will eventually reach the memory. If you recollect the memory model which we were discussing, the memory is the serialization port of the complete system. Right. So this x is going to reach here, maybe this x also reaches here.

So they will reach in the order. Suppose x equal to 1 reaches first and then x equal to 2 reaches first. So this is the hypothetical serial order which gets constructed at the memory. Okay. So x equal to 1 reaches, x equal to 2 reaches. Now what is write serialization? Suppose process P3 wants to do a read of x or we can just say that it sees x. Right.

So it is going to see x or observe x. So what is the value of x if it observes? If I have another process P4, if this P4 is just an onlooker looking at the system bus and then it just keeps on seeing what value of x is going on. Okay. So it should not happen that P3 will see first x equal to 1, x equal to 2. Whereas P4 says no I saw x equal to 2 and then I saw x equal to 1. Right. So these two different types of stories should never get constructed.

Because when P1 and P2 wrote two values, they went to the memory in a particular order. So x became 1 first and then it became 2. Hence anybody else in the system should always see x equal to 1 first and then x equal to 2. So P3 and P4 should have the same storyline that x was 1 and then x was 2. So that is the meaning of write serialization. Okay.

So writes to the same location are serialized and so any two processors have to see them in the same order. They have to be seen in the same order by everybody in the system. Okay. So we have discussed this example that if I write a 1 and then I write a 2, then everybody sees in this order and there is no process which sees the order as 2 then 1. Okay. So there should be no other process which sees the order as 2 followed by 1.

So that is the write serialization. So these are the three properties to maintain a coherent system. Now the question is about write consistency. So consistency also talks about writes, but it says when will the value of write be seen by others. Okay. So if a processor P2 changes x to some value, what is the time in the whole sequence that others will see this new value of x. So when we are talking of write consistency, we are discussing that when a write is done by a processor and a read happens to the same variable by another processor, what is the guarantee that this write will reach there because the confusion

here is that they are too close in time.

So write to x by P1. So P1 does x equal to 2 and P2 is going to do a read x, but this write and read are so close in time that P1 wrote this x equal to 2, but it has not even reached the memory. It has not reached the memory. This information has also not reached P2. So the change in P1 has not reached P2 and P2 may get a stale data value. So it is impossible to ensure that P2 will get the correct value because the change of P1 may not have also left the processor.

So P1 is just about to change x 1. It is still within its domain. It has not even reached the interconnect and when this happens and at the same time if P2 starts reading, we cannot guarantee that it will get the correct value. So if I draw the timeline like this, this is when x equal to 2 happens and this is when x equal to, x is getting read. So they are almost close to each other and hence there is no time that P1 will propagate this information to P2. So there is no time to propagate this information.

So this is what we will handle during write consistency. Okay. So of course this topic we will deal with later after we have finished all the coherence related topics. So for now until we do consistency, we need to have some assumptions on the system. Okay. So this one is just summarizing the difference that coherence defines the behavior with respect to the same memory location and consistency defines the behavior of reads and writes across other memory locations also. So here the same memory location is an important term for coherence. Okay.

So before we deal with consistency, we have to make some assumptions so that our coherent mechanism is well established and remains formal. So we are going to make two assumptions. The first assumption is that when a process does a write, until every other process has seen this write, it does not perform any other operation. Okay. And the second one is that the order of writes does not change with respect to any other instruction in the program. Okay.

So I will illustrate this. So this is the first assumption. This assumption says that writes do not complete until everybody has seen it. So, now look at the screen, you have process P1, it has done a write of b equal to 5. So it writes b equal to 5 and this b equal to 5 has to reach everybody else in the system. That is this information reaches here, maybe here, there, everywhere.

So x becoming, sorry, b becoming 5 has to be known to everybody and only then, this is when everybody knows about it, only then it can do another write. So writes do not complete and allow a next write to occur until all the processors have seen the effect of

this write. That is b equal to 5 is seen by everybody, only after everybody sees it that c equal to 5 can take place. So simple, that is the first assumption I am going to do. Okay. So this is the first assumption which says that the write of b equal to 5 should reach everybody before the P1 is permitted to do another write.

So the another write could be same location B or any other location. So b equal to 5 should reach everybody, only then b can become 6 or c can become 5 or anything else. Okay. So that is the first assumption. Right. The second assumption is, if the processor writes to location A, followed by location B, then the processor which sees the value of B will also see the value of A.

Now these are two different locations. So here processor P1 writes to b equal to 5, then it writes c equal to 2. So these are two writes and now they will reach the memory in some order. If there is another process in the system which tries to read b and c, suppose it reads c first and it sees that c is equal to 2. Now c is equal to 2 meaning processor P1 has finished both the writes and so in case this process also reads b, it should see the value of 5 and not something else. So these are writes to two different locations, right? Write to b, then write to c. Okay.

So P1, first it does write to b, then it does write to c and P2 first reads c. If it reads c, essentially it is going to read it after this, so the value it is going to get is equal to 2. Okay. So if c is equal to 2, it implies that when it does read of b, it should get the value of 5 because here the b and the c are in this order. Write to b happens before write to c and hence if you read c equal to 2, you will always read b equal to 5. Okay. Okay. So now this also puts one more extra restriction on the system which says that given this, every processor is allowed to reorder its read.

So you can read in any order, but if you are writing, your writing order must be preserved as per the program order. So if I am writing b and then c, I am not permitted to first write c and then write b. Okay. So in this example, I have taken another instance here that this is the process P3. Here first I am doing write of x and then I am doing write to y. So this order has to be maintained. So you need to maintain this whereas I am permitted to do it this way.

So I can rewrite the program and say read c and then read b. Okay. So the reads can be reordered, but the writes cannot be. So I can reorder reads but not reorder the writes. So processor does not change the order of any write with respect to any other memory location. Okay. So what does this give me? It allows the processors to reorder the reads, but writes have to finish in the program order.

That is my second assumption. So to enforce coherence or rather what is the requirements to enforce coherence? The requirement is coming out of this multiprocessor setup where I have different processors having their private caches and these caches are very important for performance because when I have a local cache, it helps me in migration and replication. Now how does it help me in migration? That is a data item, a shared data item could be located in a distant distributed memory. So it is very far from the processor and cache says that if I am going to access it, that is the locality of reference property says that once a variable is accessed, we will need the same variable and nearby variables. So when this variable is accessed, we go to the remote memory and try to bring that block into our local cache.

So the block gets migrated to my cache. Migration happens. This helps us in quick access to the variables in future and also we would not need to go over the interconnect very often and this will help us in saving the latency and the bandwidth. Okay. So migration helps me in this. And it also helps me in replication because if there is a shared variable, the shared variables could be in remote memory and every process might always go to the main memory to read them but this will incur more interconnect bandwidth and more latency. So if they individually cache them only for reading, the problem comes when we start writing. But if you only want to read, everybody can have the local copies in the local caches and it will also help them in terms of latency and saving on the interconnect bandwidth. Okay.

So why should we enforce coherence? Because the caches help me in migration by moving the data closer to improve on the latency and reduce the interconnect bandwidth. The caches also help me in replicating shared data because simultaneous readers can access the same data item kept in their local copies. Right. So only the reads of course, the writes need to be handled carefully but if everybody is reading the block, then a copy of the data can be present in its local cache. Definitely this improves the access latency and reduces the contention at the memory for the shared data because if the shared data was kept in the memory and every process went to the memory to read this variable x, there will be lot of traffic increasing the latency, increasing the latency and increasing the bandwidth demands on the memory. Okay. So replication and migration are important concepts and they have to be present to quickly access shared variables but this creates problems of cache coherence which we need to handle. Right.

So easy solution is don't cache them, other solution is let software handle but we want to handle it in hardware by maintaining coherence. For this we will need to learn different protocols and so on which we will study in the future lectures. Thank you.